

封装与接口

(OOP)

黄民烈

aihuang@tsinghua.edu.cn

<http://coai.cs.tsinghua.edu.cn/hm1/>

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 多文件编译和链接过程
- 宏定义、Make文件
- 程序命令行参数
- GDB调试工具

到目前为止，我已经完成了

- ☒ A 单文件编译、链接、运行
- ☒ B 多文件的编译、链接、运行
- ☒ C 编写了makefile，对多文件实现了自动操作
- ☐ D 我什么还没有做

提交

本讲内容提要

- 3.1 函数重载与缺省值
- 3.2 基础知识
- 3.3 类与对象
- 3.4 成员变量与成员函数
- 3.5 `private`和`public`
- 3.6 `this`指针
- 3.7 内联函数

为什么需要函数重载

■ 要表达“名一样而义不同”

- 同一任务，但输入信息的类型不同
 - sum(int a, int b);
 - sum(double a, double b);
- 同一任务，函数输入信息的存储形式不同
 - sort(const char* s);
 - sort(string str);
- 相似任务，在抽象概念层面一致
 - 如输出，有：显示到屏幕、打印到纸上、保存到文件等

第一节课：OOP的方法论；如何对客观世界进行思维抽象

函数重载

- 同一名称的函数，有两个以上不同的函数实现，被称为“函数重载”。如：

```
void print(const char* msg) { // <1>
    cout << "message: " << msg << endl;
}
void print(int score) { // <2>
    cout << "score = " << score << endl;
}
```

函数重载

- 编译器将根据函数调用语句的实际参数来决定哪一个函数被调用，如：

```
int main() {  
    print("Hello"); //调用print(const char*)  
    print(94); //调用print(int)  
    .....  
}
```

- 多个同名的函数实现之间，必须保证至少有一个函数参数的类型有区别。返回值、参数名称等不能作为区分标识。

为什么返回值不同不能作为区分

■ 假设：

```
float f(int s) {return s / 2.0;}
```

```
int f(int s) {return s * 2;}
```

■ 调用代码：

```
int main(){
```

```
    cout << f(3) << endl;
```

// 编译器应该调用哪个函数呢？

```
    return 0;
```

```
}
```


内置类型转换

- 如果函数调用语句的实参与函数定义中的形参数据类型不同，且两种数据类型在C++中可以进行自动类型转换（如int和float），则实参会被转换为形参的类型，例如：

```
#include <iostream>
using namespace std;
void print(float score) {
    cout << "score = " << score << endl;
}
int main() {
    int a = 1;
    print(a); // 此时会将a转换为float型
    return 0;
}
```

内置类型转换

- 如果形参类型为int，实参类型为float，输出结果？

```
#include <iostream>
using namespace std;
void print(int score) { cout << "score = " << score << endl; }
int main() {
    print(1.0); // score = 1
    print(1.7); // score = 1
    print(2.3); // score = 2
    print(-3.9); // score = -3, 向零取整
    return 0;
}
```

- 自动类型转换也可以通过定义的类型转换运算符来完成（之后会讲）

内置类型转换

■ 以下程序的输出结果？

```
#include <iostream>
using namespace std;
void print(int score) { cout << "int = " << score << endl; }
void print(float score) { cout << "float = " << score << endl; }
int main() {
    float a = 1.0;
    print(a); // float = 1
    return 0;
}
```

- 当函数重载时，会优先调用类型匹配的函数实现，否则才会进行类型转换

函数参数的缺省值

- 函数参数可以在定义时设置默认值（缺省值），这样在调用该函数时，若不提供相应的实参，则编译自动将相应形参设置成缺省值，如：

```
#include <iostream>
using namespace std;
void print(const char* msg = "hello") {
    cout << msg << '#';
}
int main {
    cout << "Beijing...";
    print();
    return 0;
} // 输出 Beijing...hello#
```

函数参数的缺省值

- 有缺省值的函数参数，必须是**最后一个参数**

```
void print(char* name, ✓  
           int score, ✓  
           char* msg = "pass") {  
    cout << name << ": " << score  
         << ", " << msg << endl;  
}
```

- 如果有多个带缺省值的函数参数，则这些函数参数都只能在**没有缺省值的参数**后面出现，如：

```
void print(char* name, int score=0, char*  
msg="pass")
```

函数参数的缺省值

■ 缺省值的冲突问题

- 如果因为函数缺省值，导致了函数调用的**二义性**，编译器将拒绝代码。如下面代码，会导致编译不通过。

```
void fun(int a, int b=1) {  
    cout << a + b << endl;  
}
```

```
void fun(int a) {  
    cout << a << endl;  
}
```

//测试代码

✗ fun(2); //编译器不知道该调用第一个还是第二个函数

int fun(int a) { ... }

选项中的函数不会与上述函数产生歧义的是（多选）

- A ☐ int fun(int b) { ... } X
 - B ☐ float fun(int a) { ... } X
 - C ☒ float fun(float a) { ... } ✓
 - D ☐ int fun(int a, int b=1) { ... } X
 - E ☒ int fun(int b, int a) { ... } ✓
- func 2)

提交

程序运行的结果是：

```
#include <iostream>
using namespace std;
```

```
* int fun(int a=1) { return a+1; }
* float fun(float a) { return a; }
* int fun(int a, int b) { return a+b; }
```

```
int main() {
    float a = 1.5;
    int b = 2;
    cout << fun(fun(a, b)) + fun(fun(a), b) << endl;
    return 0;
}
```

Handwritten annotations on the code:

- Under `1.5`: 1.5
- Under `fun(a, b)`: 3 (with `4` written above it)
- Under `fun(a)`: 1.5 (with `2` written above it)
- A red arrow points from the `3 to the int fun(int a, int b) function definition.`

- ☐ A 6
- ☒ B 7
- ☐ C 8
- ☐ D 9

提交

auto关键字

- C++11语法，需要std=c++11编译
- 由编译器根据上下文自动确定变量的类型，如

```
auto i = 3;    //i是int型变量
```

```
auto f = 4.0f;    //f是float型变量
```

```
auto a('c'); //a是char型变量
```

```
auto b = a; //b是char型变量
```

```
auto *x = new auto(3); //x是int*
```

 Δ

auto关键字

■ 追踪返回类型的函数

- 可以将函数返回类型的声明信息放到函数参数列表的后面进行声明，如：

- 普通函数声明形式

int func(char* ptr, int val);

- 追踪返回类型的函数声明形式

auto func(char* ptr, int val) -> int;

- 追踪返回类型在原本函数返回值的位置使用auto关键字

auto关键字

■ **auto** 变量必须在编译期确定其类型

■ **auto** 变量必须在定义时初始化

- auto a; // 错误
- auto b4 = 10, b5 = 20.0, b6 = 'a';
// 错误, 没有推导为同一类型

■ 参数不能被声明为auto

- void func(auto a) {...} // 错误

sizeof(int)

■ **auto**并不是一个真正的类型。不能使用一些以类型为操作数的操作符，如sizeof或者typeid。

- cout << sizeof(auto) << endl; // 错误

decltype

■ decltype

- decltype可以对变量或表达式结果的类型进行推导
- 重用匿名类型 *Test*

```
struct { int d ;  
        double b;  
} anon_s; // 没有名字的结构体，定义了一个变量  
  
int main() {  
    decltype(anon_s) as ;  
    // 定义了一个上面匿名的结构体...  
}
```

decltype

■ decltype

- decltype可以对变量或表达式结果的类型进行推导。

```
struct { char name[17]; } anon_u;
```

```
struct {  
    int d;
```

```
    decltype(anon_u) id;
```

```
} anon_s[100]; // 匿名的struct数组
```

```
int main() {
```

```
    decltype(anon_s) as; // 注意变量as的类型:数组
```

```
    cin >> as[0].id.name;
```

```
    ...
```

```
}
```

auto+decltype

■ 结合auto和decltype，自动追踪返回类型

- 可以推导返回类型 (C++11)

```
auto func(int x, int y) -> decltype(x+y)  
{  
    return x+y;  
}
```

✗

- C++14中不再需要显式指定返回类型

```
auto func(int x, int y)  
{  
    return x+y; ✗  
}
```

WHY auto?

- 用于代替冗长复杂、变量使用范围专一的变量声明。

```
std::vector<std::string> vs;
```

```
for (std::vector<std::string>::iterator i =  
vs.begin(); i != vs.end(); i++) △  
{ //..... }
```

```
std::vector<std::string> vs;  
for (auto i = vs.begin(); i != vs.end(); i++)  
{ //..... }
```

WHY auto?

- 在定义模板函数时，用于声明依赖模板参数的变量类型。

```
template <typename _Tx, typename _Ty>
void Multiply(_Tx x, _Ty y)
{
    auto v = x*y; //临时变量
    std::cout << v;
}
```

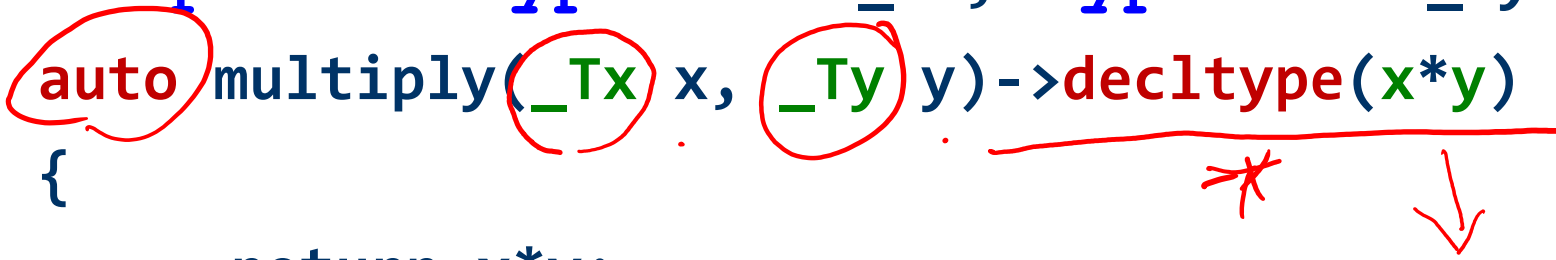
//使用时

```
Multiply(2, 3); //Multiply(int, int)
Multiply(2, 3.3); //Multiply(int, double)
```


WHY auto?

- 结合auto和decltype, 自动追踪返回类型

```
template <typename _Tx, typename _Ty>  
auto multiply(_Tx x, _Ty y) -> decltype(x*y)  
{  
    return x*y;  
}
```



//使用时

int double

```
auto a = multiply(2, 3.3); //a=6.6
```

内存申请与释放

■ 内存的动态申请与释放

- 指针变量所指内存可以通过new/delete运算符在程序运行时动态生成和删除，如：

~~* ptr = 10;~~

int * ptr = new int(10); // 单个变量 ✓

int * array = new int[10]; // 10元素数组

delete ptr; // 删除指针变量所指单个内存单元

~~delete~~ array; // 删除多个单元组成的内存块

零指针

■ 0、NULL、nullptr

- 在C++中，NULL被定义为0

```
#ifdef __cplusplus  
#define NULL 0  
#else  
#define NULL ((void *)0)  
#endif
```

- 在C++11之前，可以使用NULL或者0表示空指针。
- 这么做有什么问题？

零指针

■ 减少NULL的使用 (c++11之前)

- 定义一个函数，并对它进行调用

```
void f(int x, int y) {...}  
f(2, 0);
```

- 如果我们想对这个函数进行重载，并传入一个空指针作为参数

```
void f(int x, double *y) {...}
```

⊗ f(2, NULL); ☹ // 实际调用的不是我们所期望的
f(2, static_cast<double *>(0)); ☺

- 当我们使用NULL表示空指针时，容易忽略它同时是一个int型常量。 *

零指针

■ nullptr 的引入 (c++11)

- nullptr 表示严格意义上的空指针。
- 此时再执行之前的代码，不会产生错误。

```
✓ void f(int x, int y) {  
    cout<<"int"<<endl;  
}
```

```
✓ void f(int x, double *y) {  
    cout<<"pointer"<<endl;  
}
```

```
f(2, nullptr); *
```

Output: pointer

For循环

■ 基于范围的for循环

- 在循环头的圆括号中，由冒号":"分为两部分，第一部分是用于迭代的变量，第二部分则表示将被迭代的范围。
如：

```
#include <iostream>
using namespace std;
int main() {
    int arr[3] = {1, 3, 9};
    for (int e : arr) // auto e:arr 也可以
        cout << e << endl;
    return 0;
}
```

以下语句能够编译的有（多选）

☒ A `auto x = new int[10];` ✓

☒ B `auto *x = new int[10];` ✓

☒ C `auto x = "123";` ✓

☒ D `auto *x = "123";` ✓

提交

以下能够在C++14下正确编译的有（多选）

☒ A `for(auto x : "123") { ... }`

☐ B `void f(auto x) { ... }`

☒ C `auto f(int x) { ... }`

☐ D `auto f(auto x) { ... }`

提交

OOP从认识“对象”开始.....

■ 对象，是对现实世界中实际存在事物的抽象描述，它可以是有形的，也可以是无形的

- (1) 对象具有自己的静态特征和动态特征
 - 静态特征 — 可以用某种数据来描述的属性；
 - 动态特征 — 对象表现的行为或具有的功能。
- (2) 对象是由一组属性数据和对这些数据进行特定操作的一组服务所构成的“结合体”（概念）。

■ 封装 = {属性/数据, 服务/函数}

封装的“装”——数据抽象

■ 数据+函数——从设计思想上看

- 封装（数据+函数）是OOP的基本特征
- 从只关心数值的存储与表示，到既考虑内容，又考虑语义（数据支持的计算或操作），形成了对“数据”概念的更本质认识，这种思维过程称为“数据抽象”。

用户定义类型——类class

■ class 用户自定义的类型

- 包含函数与数据的特殊“结构体”，用于扩充C++语言的类型体系
- 类中包含的函数，称为“成员函数”
- 包含的数据，称为“成员变量”

■ 成员函数必须在类内声明，但定义（实现）可以在类内或者类外。

■ 类= “属性/数据” + “服务/函数”



在头文件中声明类class

```
// matrix.h  
#ifndef MATRIX_H  
#define MATRIX_H  
  
class Matrix {  
    int data[6][6]; // 成员变量  
public:  
    void fill(char dir); // 成员函数 (声明)  
};  
  
#endif
```

在实现文件中定义成员函数

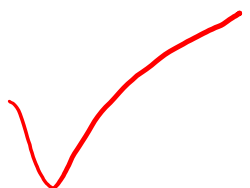
```
// matrix.cpp  
#include "matrix.h"  
  
void Matrix::fill(char dir) // 类外需要类名限定  
{  
    ... // 函数实现  
}
```

定义


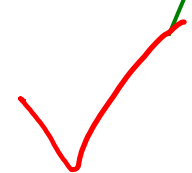
- 通常，类的**声明**放在头文件中，而类的成员函数**实现**（也叫**定义**）则放在实现文件中。
- 为了便于管理和代码复用，一般是将不同的类分别保存为不同的头文件和实现文件。

成员函数的两种定义方式

```
class Matrix {  
public:  
    void fill(char dir) {  
        ...; //在类内定义成员函数  
    }  
    ...  
}; // <1>
```



```
void Matrix::fill(char dir) {  
    ... ; //在类外定义成员函数  
} // <2>
```



- 为了方便解决依赖关系，复杂的成员函数声明和定义一般是分离的，很少使用类内定义的方式。

类成员的访问权限

- 类的成员（数据、函数）可以根据需要分成组，不同组设置不同的访问权限

■ public

- 被public修饰的成员可以在类外访问。

■ private

- 默认权限
- 被private修饰的成员不允许在类外访问。

■ protected

- 以后介绍。

用户定义类型——类class

- 定义类后，可以像语言内建的类型一样，用类来定义变量，该变量通常被称为“对象”
int, double, char
Matrix a;
- 通过“对象名.成员名”的形式，可以使用对象的数据成员，或调用对象的成员函数。在类外使用时仅限于访问public权限的成员
- 同样，可以使用“对象指针->成员名”形式访问数据成员或成员函数。在类外也只限于访问public权限的成员。

类成员的访问权限

```
// matrix.h
```

```
class Matrix {
```

```
public:
```

```
    void fill(char dir);
```

```
private:
```

```
    int data[6][6];
```

```
}; // <1>
```

```
//或者
```

```
class Matrix {
```

```
    int data[6][6]; // class中成员的缺省属性为private
```

```
public:
```

```
    void fill(char dir);
```

```
}; // <2>
```

类成员的访问权限

```
// main.cpp
```

```
#include "matrix.h"
```

```
int main()
```

```
{
```

```
    Matrix obj;
```

```
    obj.fill('u');
```

```
    obj.data[1][1] = 23;
```

```
    return 0;
```

```
}
```

```
// Matrix类的声明
```

```
// 定义变量（对象）
```

```
// 访问公有成员
```

```
// ERROR!
```

- 不允许在类外(非该类的成员函数)操作访问对象的私有成员和保护成员，只能访问它的公有属性的成员（函数、数据）。

类成员的访问权限

```
// matrix.h
class Matrix {
private:
    int data[6][6];
    void add(Matrix a);
public:
    void fill(char
dir);
};
```

私有

```
// matrix.cpp
#include "matrix.h"

void Matrix::add(Matrix a) {
    for(int i=0; i<6; i++)
        for(int j=0; j<6; j++)
        {
            data[i][j] += a.data[i][j]
            // 可以在类内用"."操作访问同一类下的私有成员
        }
```

this指针

Matrix a;

- 所有成员函数的参数中，隐含着一个指向当前对象的指针变量，其名称为this

```
class Matrix {  
public:
```

这也是成员函数与普通函数的重要区别

```
    void fill(char dir) { // <1> 在类内定义成员函数
```

```
        ...
```

```
        this->data[0][0] = 1; // 等价于 data[0][0] = 1;
```

```
    }
```

```
    ...
```

```
};
```

```
void Matrix::fill(char dir) { // <2> 在类外定义成员函数
```

```
    this->data[0][0] = 1; // 等价于 data[0][0] = 1;
```

```
    ... ;
```

```
}
```

有如下程序：

```
#include <iostream>
using namespace std;
class A {
private:
    int a;
    void f(int i=2) { a = i; }
public:
    void f(int i, int j=2) { a = i + j; }
    int get_a() { return a; }
};

int main() {
    A aa;
    aa.f(1);
    cout << aa.get_a() << endl;
    return 0;
}
```

- ☐ A 该程序正常运行，输出结果为1
- ☐ B 该程序正常运行，输出结果为3
- ☒ C 该程序无法通过编译

提交

```
class P {  
private:  
    int data = 1;  
    void add(P a);  
public:  
    void add(int i) { data += i; }  
};  
void P::add(P a) { data += a.data; } // A  
  
class Q {  
private:  
    void add(P a) { data += a.data; } // B  
public:  
    int data = 2;  
};  
int main() {  
    P a, b; Q c;  
    int d = c.data; // C  
    a.add(b); // D  
    a.add(d); // E  
    return 0;  
}
```

左侧的代码中
哪些操作是合法的

- ☒ A ✓
- ☐ B ✗
- ☒ C ✓
- ☐ D ✗
- ☒ E ✓

提交

```

class P {
private:
    int data = 1;
    void add(P a);
public:
    void add(int i) { data += i; }
};
void P::add(P a) { data += a.data; } // A: 在P类私有函数
内访问同类P对象的私有成员，类内访问

```

```

class Q {
private:
    void add(P a) { data += a.data; } //B: Q类函数内访问P
类对象的私有成员

```

```

public:
    int data = 2;
};
int main() {
    P a, b; Q c;
    int d = c.data; // C: Q类对象的公有数据成员
    a.add(b); // D: 在main函数内，P类对象的私有函数
    a.add(d); // E: P类对象的公有函数
    return 0;
}

```

思考

Test a, b,

a+b?

a * b?

■ 基本类型和自定义类型的差别是什么?

■ 基本类型: `int`, `long`, `char`, `double`, `float`

• 数据是什么, 操作什么?

+ - * /

■ 自定义类型

• 数据是什么? 操作是什么?

char "+"

```
class Test {  
    int data[100];  
public:  
    void setdata(const int*);  
    const int* getdata();  
    void operation1(int);  
};
```

int

数据

操作

内联函数

■ 为什么用内联函数

- 考虑一个很常用的函数

```
int max(int a, int b) {  
    return a > b ? a : b;  
}
```

- 函数调用要进行一系列准备和后处理工作(压栈、跳转、退栈、返回等)，所以函数调用是一个比较慢的过程。
- 如果大量调用**max**函数，会拖慢程序。

内联函数

■ 为什么用内联函数

- 考虑一个很常用的函数

```
int max(int a, int b) {  
    return a > b ? a : b;  
}
```

- 比较下面两种实现方式，函数比等价的表达式要慢得多！

✓ `cout << max(a, b) << endl;`

✓ `cout << (a > b ? a : b) << endl;` ✗ ✗ ✗

内联函数

■ 为什么用内联函数

- 使用内联函数，编译器自动产生等价的表达式。

```
inline int max(int a, int b) {  
    return a > b ? a : b;  
}  
cout << max(a, b) << endl;
```

- 上述代码等价于

```
cout << (a > b ? a : b) << endl;
```

内联函数

■ 内联函数和宏定义的区别

- 宏定义

```
#define MAX(a, b) (a) > (b) ? (a) : (b)
```

```
cout << MAX(a, b) << endl;
```

- 编译器会将所有宏定义的代码，直接拷贝到被调用的地方。上面对于**MAX**的调用，经过编译预处理器后，和下面代码完全等价。

```
#define MAX(a, b) (a) > (b) ? (a) : (b)
```

```
cout << (a) > (b) ? (a) : (b) << end;
```

内联函数

■ 内联函数和宏定义的区别

- 宏代码容易出错，编译预处理器在拷贝代码时，可能产生意想不到的边界效应。

```
#define MAX(a, b) (a) > (b) ? (a) : (b)
```

```
cout << MAX(a, b) + 2 << endl;
```

```
cout << (a) > (b) ? (a) : (b) + 2 << end; //编译预处理器结果☹
```

- 上述代码更改如下可正常工作。

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

```
cout << MAX(a, b) + 2 << endl;
```

```
cout << ((a) > (b) ? (a) : (b)) + 2 << end; // 编译预处理器结果☺
```

- 这样的宏代码万无一失了吗？

内联函数

■ 内联函数和宏定义的区别

- 显然不是万无一失的，例如下面的这种情况。

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

```
cout << MAX(a++, b) + 2 << endl;     加1次
```

```
cout << ((a++) > (b) ? (a++) : (b)) + 2 << end; // a被两次求值☹
```

- 因为宏代码是直接拷贝到指定位置的，很多缺陷不可避免。
- 内联函数的好处显而易见！

内联函数

■ 内联函数和宏定义的区别

- 内联函数可以执行类型检查，进行编译期错误检查。
- 内联函数可调试，而宏定义的函数不可调试。
 - 在Debug版本，内联函数没有真正内联，而是和一般函数一样，因此在该阶段可以被调试。
 - 在Release版本，内联函数实现了真正的内联，增加执行效率。
- 宏定义的函数无法操作私有数据成员。
- 宏使用的最常见场景：字符串定义、字符串拼接、标志粘贴（教材第9章p221~222）
- 再次强调
 - 宏定义只是拷贝代码到被调用的地方。
 - 内联函数生成的是，和函数等价的表达式。

内联函数

■ 内联函数的注意事项

- 避免对大段代码使用内联修饰符。
 - 内联修饰符相当于把该函数在所有被调用的地方拷贝了一份，所以大段代码的内联修饰会增加负担。（代码膨胀过大）
- 避免对包含循环或者复杂控制结构的函数使用内联定义。
 - 因为内联函数优化的，只是在函数调用的时候，会产生压栈、跳转、退栈和返回等操作。所以如果函数内部执行代码的时间比函数调用的时间长得多，优化几乎可以忽略。
- 避免将内联函数的声明和定义分开
 - 编译器编译时需要得到内联函数的实现，因此多文件编译时内联函数先需要将实现写在头文件中，否则无法实现内联效果。
- 定义在类声明中的函数默认为内联函数。
- 一般构造函数、析构函数都被定义为内联函数。

内联函数

```
//test.h
class Test {
    int* data;
public:
    void setdata(const int* d) {data = d; } //内联函数 ✓
    const int* getdata() {return this->data;} //内联函数
    void operation1(int);
    Test(int i){ //
        if (i>0)
            data = new int[i];
        else
            data = nullptr;
    } //内联函数
    ~Test(){delete[] data;} //内联函数
};
```

内联函数

■ 内联函数的注意事项

- 内联修饰符更像是建议而不是命令。✓
- 编译器“有权”拒绝不合理的请求，例如编译器认为某个函数不值得内联，就会忽略内联修饰符。✓
- 编译器会对一些没有内联修饰符的函数，自行判断可否转化为内联函数，一般会选择短小的函数。✓

选择正确的说法（多选题）

A

使用内联函数，可以减小程序代码大小，
但使程序执行的速度减慢

B

用inline修饰的函数不建议使用循环语句和switch语句

C

编译器可以将递归的函数内联

D

编译器内联时是将该函数的目标代码插入
每个调用该函数的地方

提交

课后阅读

■ 《C++编程思想》

- 隐藏实现, p142-p151
- 函数重载与默认参数, p171-p180
- 内联函数, p207-p220

■ 如何在C++中打印变量类型

- <https://stackoverflow.com/questions/81870/is-it-possible-to-print-a-variables-type-in-standard-c>

结 束